OR					
9	a)	Explain the usage of try, catch, throw, throws and finally keywords in exception handling. Give simple example.	L2	CO3	5 M
	b)	Illustrate the difference between byte streams and character streams in Java. Draw the stream hierarchies.	L3	CO3	5 M
UNIT-V					
10	a)	Explain how thread priority is set and used in Java with example program.	L4	CO4	5 M
	b)	Illustrate the use of HashSet class in collection framework with an example program.	L3	CO4	5 M
OR					
11	a)	Analyze different procedures for creating a thread in Java. Explain any one mechanism with example program.	L4	CO4	5 M
	b)	Define List and differentiate ArrayList, LinkedList.	L2	CO4	5 M

Code: 23CS3302, 23IT3302, 23AM3302, 23DS3302

II B.Tech - I Semester – Regular Examinations - DECEMBER 2024

OBJECT ORIENTED PROGRAMMING THROUGH JAVA

(Common for CSE, IT, AIML, DS)

Duration: 3 hours Max. Marks: 70

Note: 1. This question paper contains two Parts A and B.
2. Part-A contains 10 short answer questions. Each Question carries 2 Marks.
3. Part-B contains 5 essay questions with an internal choice from each unit. Each Question carries 10 marks.
4. All parts of Question paper must be answered in one place.

BL – Blooms Level CO – Course Outcome

PART – A

		BL	CO
1.a)	Develop a program to implement Command Line Arguments in Java.	L3	CO1
1.b)	Explain the purpose of type casting in Java.	L2	CO1
1.c)	How do you access private members of a class? Explain.	L2	CO2
1.d)	Explain the methods used for searching with-in string in Java.	L2	CO2
1.e)	Explain how you can dynamically change the size of an array.	L2	CO2
1.f)	Define abstract class. Differentiate with concrete class.	L2	CO2
1.g)	Identify the differences between auto-boxing and auto-unboxing.	L2	CO3
1.h)	How does the Scanner class facilitate input	L2	CO3

	operations in Java? Explain.		
1.i)	Identify the main states in the Java thread life cycle.	L2	CO4
1.j)	What is the purpose of Collection Framework in Java?	L1	CO4

PART – B

			BL	CO	Max. Marks
		UNIT-I			
2	a)	List and explain different data types supported by Java. Give suitable example program.	L2	CO1	5 M
	b)	Write a Java program to calculate the tax on a salary. The program should prompt the user to enter their annual salary. Use the following tax brackets: Sal< 2,50,000 rupees – No tax Sal > 2,50,001 and < 5,00,000 - 10% tax Sal> 5,00,000 – 5% tax Display the tax amount based on the entered salary.	L3	CO1	5 M
		OR			
3		Explain the different types of operators in Java with examples.	L2	CO1	10 M
		UNIT-II			
4	a)	Differentiate constructor overloading and method overloading. Give suitable examples.	L3	CO2	5 M

	b)	Explain how you can modify a string with example program.	L2	CO2	5 M
		OR			
5	a)	Describe the process of declaring and initializing class and objects in Java with suitable example.	L2	CO2	5 M
	b)	Construct a Java program to differentiate passing arguments by Value and by Reference.	L3	CO2	5 M
		UNIT-III			
6	a)	List and explain different operations that can be performed on Array elements.	L2	CO2	5 M
	b)	What is inheritance? Explain different types of inheritance techniques which are supported by Java.	L2	CO2	5 M
		OR			
7	a)	Explain the following: i) final keyword in inheritance ii) Vector	L2	CO2	5 M
	b)	Discuss the concepts of default and static methods in Interface.	L2	CO2	5 M
		UNIT-IV			
8	a)	How do different access control specifiers control access to class members across different packages? Explain.	L2	CO3	5 M
	b)	Differentiate between checked and unchecked exceptions. Give suitable examples.	L3	CO3	5 M

Code:23CS3302, 23IT3302, 23AM3302, 23DS3302 PVP23

II B.Tech. - I Semester-Regular Examinations - DECEMBER 2024

OBJECT ORIENTED PROGRAMMING THROUGH JAVA

(Common to CSE, IT, AIML, DS)

Duration:3 hours Maximum Marks : 70

Note: 1. This Question paper contains Part A and B

2. Part A Contains 10 short answer Questions. Each question carries 2 Marks
3. Part B contains 5 essay Questions with an internal choice from each unit. Each question carries 10 Marks.
4. All parts of the question paper must be answered in one place.

BL – Blooms Level CO – Course Outcomes

PART – A

1.a) Develop a program to implement Command Line arguments in Java. L3 CO1

Any Simple Java Program and show how to give values at the time of running a program at command prompt - 2M

1.b) Explain the purpose of type casting in Java. L2 CO1

Implicit – 1 M Explict – 1M

1.c) How do you access private members of a class? Explain. L2 CO2

Accessing private members of a class i.e. either variable or method – 2M

1.d) Explain the methods used for searching with-in String in Java. L2 CO2

Atleast 2 methods with syntax and example – 2M

1.e) Explain how you can dynamically change the size of an Array. L2 CO2

Dynamically Array Size change explanation – 2M

1.f) Define abstract class. Differentiate with concrete class. L2 CO2

Abstract class definition – 1M , Difference with concrete class – 1M

1.g) Identify the differences between auto-boxing and auto-unboxing. L2 CO3

Differences between **auto-boxing and auto-unboxing.**

1.h) How does the Scanner class facilitate input operations in Java? Explain. L2 CO3

Taking input for different primary data types using scanner class methods – 2M

1.i) Identify the main states in the Java thread life cycle. L2 CO4

List of thread classes – 2M

1.j) What is the purpose of Collection Framework in Java? L2 CO4

Collection framework purpose – 2M

PART – B

UNIT – I

2. a) List and Explain different data types supported by Java. Give suitable example Program. L2 CO1 5 M

Data types list – 2M
Example Program – 3M

b) Write a java program to calculate the tax on a salary. The program should prompt the user to enter their annual salary.

Use the following tax brackets:

Sal<2,50,00 rupees – No tax

Sal>2,50,001 and Sal<5,00,000 - 10% tax

Sal > 5,00,000 – 5% tax.

Display the tax amount based on the entered salary. L3 CO1 5M

Input – 1M

Conditions – 3M Display/Output – 1M

OR

3. Explain the different types of operators in Java with examples. L2 CO1 10M

List of operators – 3 M

Example programs – 7M

UNIT – II

4. a) Differentiate constructor overloading and method overloading. Give suitable examples. L3 CO2 5M

Differences between constructor and method overloading – 3M

Example – 2M

b) Explain how you can modify string with example Program. **L2 CO2 5M**

5 methods with syntax and example – 5 M

OR

5. **a) Describe the process of declaring and initializing class and objects in Java with suitable example.** **L2 CO2 5M**

Declaring and initializing class = 2 ½ M

Declaring and initializing object = 2 ½ M

b) Construct a Java Program to differentiate passing arguments by value and by Reference. **L3 CO2 5M**

Passing arguments by values – 2 ½ M

Passing arguments by Reference – 2 ½ M

UNIT – III

6. **a) List and explain different operations that can be performed on Array elements.** **L2 CO2 5M**

Array operations with syntax and example – 5M

b) What is inheritance? Explain different types of inheritance techniques which are supported by Java. **L2 CO2 5M**

Inheritance definition – 1M

3 types explanation – 4M

OR

7. **a) Explain the following** **L2 CO2 5M**
   i) final keyword in inheritance **ii) vector**

final keyword – 2 ½ M

vector – 2 ½ M

b) Discuss the concepts of default and static methods in interface. **L2 CO2 5M**

default method in interfaces concept – 2 ½ M

static method in interfaces concept – 2 ½ M

UNIT – IV

8. **a) How do different access control specifiers control access to class members**

across different packages? Explain. **L2 CO3 5M**

Access control specifiers accessing in classes and packages explanation – 5M

b) Differentiate between checked and unchecked exceptions. Give suitable examples. **L3 CO3 5M**

Differences between checked and unchecked exceptions with example – 5M

OR

9. a) Explain the usage of try, catch, throw, throws and finally keywords in exception handling. Give simple example. **L2 CO3 5M**

each keyword with example – 5M

b) Illustrate the difference between byte stream and character streams in Java. Draw the stream hierarchies. **L3 CO3 5M**

Differences between byte stream and character stream – 4M

Stream hierarchies – 1 M

UNIT – V

10. a) Explain how thread priority is set and used in Java with example program. **L4 CO4 5M**

Thready priority explanation – 2M Example Program - 3M

b) Illustrate the use of HashSet class in collection framework with an example program. **L3 CO4 5M**

usage of HashSet in java – 2M Example Program – 3M

OR

11. a) Analyze different procedures for creating a thread in Java. Explain any one mechanism with example program. **L4 CO4 5M**

Different ways of Creation of a thread in java explanation – 2 M

Example program – 3M

b) Define List and differentiate ArrayList, Linked List. **L2 CO4 5M**

List definition – 1 M

Differentiate with Array List – 2 M, with Linked List – 2M

Code:23CS3302, 23IT3302, 23AM3302, 23DS3302 PVP23

II B.Tech. - I Semester-Regular Examinations - DECEMBER 2024

OBJECT ORIENTED PROGRAMMING THROUGH JAVA

(Common to CSE, IT, AIML, DS)

Duration:3 hours Maximum Marks : 70

Note: 1. This Question paper contains Part A and B

2. Part A Contains 10 short answer Questions. Each question carries 2 Marks
3. Part B contains 5 essay Questions with an internal choice from each unit. Each question carries 10 Marks.
4. All parts of the question paper must be answered in one place.

BL – Blooms Level CO – Course Outcomes

PART – A

1.a) Develop a program to implement Command Line arguments in Java. L3 CO1

```
public class AddNumbers {
public static void main(String[] args)
// Convert the string arguments to integers
int num1 = Integer.parseInt(args[0]);
int num2 = Integer.parseInt(args[1]);
int result = num1 + num2;
System.out.println("The sum of " + num1 + " and " + num2 + " is: " + result);
}
}
```

1.b) Explain the purpose of type casting in Java. L2 CO1

In Java, type casting is the process of converting a variable from one type to another.

There are two types of casting

1. **Implicit Type Casting (Automatic Casting)**: This occurs when you assign a value of a smaller or narrower data type to a larger or wider data type. Java performs this conversion automatically because it is safe and does not result in data loss.

   Example:
   ```
   // Implicit Type Casting (Automatic Casting)
   int intValue = 100;
   ```

double doubleValue = intValue; // int is automatically cast to double

2. **Explicit Type Casting (Manual Casting)**: This is necessary when you want to convert a value from a larger or wider data type to a smaller or narrower data type. Since this conversion might result in data loss, you need to specify it explicitly.
   // Explicit Type Casting (Manual Casting)
   double doubleValue2 = 9.78;
   int intValue2 = (int) doubleValue2; // double is manually cast to int

1.c) How do you access private members of a class? Explain. **L2 CO2**

When we use a private access specifier, the method is accessible only in the classes in which it is defined.

```
class A{
private int data=40;
private void msg()
{  System.out.println("Hello java");}
}

public class Simple{
 public static void main(String args[]){
  A obj=new A();
  System.out.println(obj.data);//Compile Time Error
  obj.msg();//Compile Time Error
  }
}
```

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is a compile-time error.

1.d) Explain the methods used for searching with-in String in Java. **L2 CO2**

contains() : Checks if a substring exists in the string.
indexOf() : Returns the index of the first occurrence of the specified substring or character. Returns -1 if not found.

charAt(int index) : The charAt(int index) method in Java is used to retrieve the character at a specific position (index) in a string.

lastIndexOf(char c) : The lastIndexOf() method starts searching backward from the end of the string and returns the index of specified characters whenever it is encountered.

lastIndexOf(char c, int fromIndex) : It starts searching backward from the specified index in the string. And returns the corresponding index when the specified character is encountered otherwise returns -1.
Note: The returned index must be less than or equal to the specified index.

1.e) Explain how you can dynamically change the size of an Array. **L2 CO2**

The number of elements (size) of the array may change during the execution of the program.

In Java, change the number of elements by dynamically retaining the array name. In this process, the old array is destroyed along with the values of elements.

Example: int [ ] num = new int [5];

num = new int [10];

(or)

To dynamically change the size of an array, we can create a new array with a different capacity and copy the elements from the old array to the new one using the **Arrays.copyOf()** method. We can then delete the old array and replace it with the new one.

Example :

```
int[] oldArray = {1, 2, 3};
int[] newArray = Arrays.copyOf(oldArray, 5); // New size is 5
newArray[3] = 4;
newArray[4] = 5;
System.out.println(Arrays.toString(newArray));
```

1.f) Define abstract class. Differentiate with concrete class. **L2 CO2**

Abstract Class: An **abstract class** in Java is a class that cannot be instantiated directly and is intended to be extended by other classes. It serves as a blueprint for other classes, providing a way to define methods that must be implemented by subclasses, as well as methods that can have default behaviour. We cannot create an object of an abstract class. It is meant to be inherited by other classes. An abstract class can contain both abstract methods (without implementation) and non-abstract methods (with implementation).

Concrete Class: A concrete class is a class that can be instantiated and provides implementation for all its methods. It may or may not inherit from an abstract class.

Difference:

Abstract Class	Concrete Class
1. Cannot be instantiated directly.	1. Can be instantiated.
2. May contain abstract methods (without implementation).	2. Does not contain abstract methods.

Abstract Class	Concrete Class
3. Meant to be inherited by subclasses.	3. Can be inherited but doesn't need to be.
4. Provides a template for other classes.	4. Fully implements methods and can be used directly.

1.g) Identify the differences between auto-boxing and auto-unboxing. **L2 CO3**

Autoboxing	Unboxing
The automatic conversion of a primitive type to its corresponding wrapper class.	The automatic conversion of a wrapper class object to its corresponding primitive type.
Primitive values are converted into wrapper objects.	Wrapper objects are converted into primitive values.
Occurs when a primitive type is assigned to a wrapper class.	Occurs when a wrapper class object is assigned to a primitive type.

public class autoboxing {
public static void main(String args[]) {
// Converting int into Integer (Manual)
int a = 20;
// Converting int into Integer using valueOf()
Integer i = Integer.valueOf(a);
// Autoboxing (Automatic)
Integer j = a; **// Autoboxing: compiler internally uses Integer.valueOf(a)**
System.out.println(a + " " + i + " " + j);
}
}
	public class autounboxing {
	public static void main(String args[]) {
	// Converting Integer to int (Manual)
	// Creates an Integer object
	Integer a = new Integer(3);
	int i = a.intValue(); **// Converting Integer to int using intValue()**
	// Unboxing (Automatic)
	int j = a; **// Unboxing: compiler internally uses a.intValue()**
	System.out.println(a + " " + i + " " + j);
	}
	}

1.h) How does the Scanner class facilitate input operations in Java? Explain. **L2 CO3**

The Scanner class in Java facilitates input operations by providing methods to read different types of data from various input sources such as the keyboard, files, or streams.
Scanner class is in java.util package.
It allows the user to read data like strings, integers, and other primitive types using methods like nextInt(), nextLine(), nextDouble(), etc.

- nextLine() reads a full line of text as a String.
- nextInt() reads an integer.
- nextDouble() reads a decimal number (double).

1.i) Identify the main states in the Java thread life cycle. **L2 CO4**

- **Newborn State**: The thread is created but not yet started.

- **Runnable State**: The thread is ready to run and waiting for CPU time.
- **Running State**: The thread is currently executing its task.
- **Blocked State**: The thread is waiting to acquire a resource (e.g., locked by another thread).
- **Dead State**: The thread has completed its execution or terminated.

1.j) What is the purpose of Collection Framework in Java? **L2 CO4**

The Java Collections Framework is a set of classes and interfaces that help represent and manipulate collections of objects.
The purpose of the Collection Framework in Java is to provide a unified architecture for storing, managing, and manipulating groups of objects in a standardized and efficient manner. It offers a set of interfaces, implementations, and algorithms for handling data structures, such as lists, sets, and queues, ensuring flexibility, reusability, and performance optimization in Java programs.

PART – B

UNIT – I

2. a) List and Explain different data types supported by Java. Give suitable example Program. **L2 CO1 5 M**

Data types specify the different sizes and values that can be stored in the variable.

I. Primitive datatypes: Primitive Data Types are predefined and available within the Java language.

1.**Byte**: Represents an 8-bit signed integer. Range: -128 to 127 Default Value: 0

2.**short**: Represents a 16-bit signed integer. Range: -32,768 to 32,767Default Value: 0

3.**int:** Range: -2^31 to 2^31-1 Default Value: 0

4.**long**: Represents a 64-bit signed integer. Range: -2^63 to 2^63-1 Default Value: 0L

5.**float**: Represents a single-precision 32-bit IEEE 754 floating-point. Range: Approximately ±3.40282347E+38F (6-7 significant decimal digits) Default Value: 0.0f 6.**double:** Represents a double-precision 64-bit IEEE 754 floating-point. Range: Approximately ±1.79769313486231570E+308 (15 decimal digits) Default Value: 0

7.**char:** Represents a single 16-bit Unicode character. Range: 0 to 65,535 Default Value: '\u0000'

8.**boolean:** Represents a value of true or false. Range: true or false Default Value: false

II. Non-primitive Datatypes:

1. **String:** Represents a sequence of characters.
2. **Arrays:** Represents a collection of elements of the same type.
3. **Class :** Represents a template to the data which consists of member variables and methods.
4. **Objects:** Represents instances of classes.

5. **Interfaces:** It is similar to a class however the only difference is that its methods are abstract by default i.e. they do not have body. An interface has only the final variables and method declarations. It is also called a fully abstract class.

```
public class DataTypeDemo {
public static void main(String[] args) {
byte byteValue = 10;
short shortValue = 1000;
int intValue = 100000;
long longValue = 10000000000L;
float floatValue = 10.5f;
double doubleValue = 20.99;
char charValue = 'A';
boolean booleanValue = true;
String stringValue = "Hello, World!";
int[] intArray = {1, 2, 3, 4, 5};
System.out.println("Primitive Data Types:");
System.out.println("byteValue: " + byteValue);
System.out.println("shortValue: " + shortValue);
System.out.println("intValue: " + intValue);
System.out.println("longValue: " + longValue);
System.out.println("floatValue: " + floatValue);
System.out.println("doubleValue: " + doubleValue);
System.out.println("charValue: " + charValue);
System.out.println("booleanValue: " + booleanValue);
System.out.println("\nReference Data Types:");
System.out.println("stringValue: " + stringValue);
System.out.print("intArray: ");
for (int i = 0; i < intArray.length; i++) {
System.out.print(intArray[i] + " ");
}
System.out.println();
}
}
```

b) Write a java program to calculate the tax on a salary. The program should prompt the user to enter their annual salary.

Use the following tax brackets:

Sal<2,50,00 rupees – No tax

Sal>2,50,001 and Sal<5,00,000 - 10% tax

Sal > 5,00,000 – 5% tax.

Display the tax amount based on the entered salary. L3 CO1 5M

```
import java.util.Scanner;
public class SalaryTaxCalculator {
  public static void main(String[] args) {
    // Create a Scanner object for user input
    Scanner sc = new Scanner(System.in);
    // Prompt the user to enter their annual salary
    System.out.print("Enter your annual salary: ");
    double salary = sc.nextDouble();
    // Variable to store the calculated tax
    double tax = 0;
    // Calculate tax based on salary brackets
    if (salary <= 250000) {
      tax = 0; // No tax for salaries <= 2,50,000
    }
      else if (salary > 250000 && salary <= 500000) {
      tax = (salary - 250000) * 0.10; // 10% tax for the portion above 2,50,000
    }
      else if (salary > 500000) {
      // 10% tax for the portion between 2,50,001 and 5,00,000 and 5% for the portion above
      5,00,000
          tax = (salary - 500000) * 0.05 + (250000 * 0.10);
    }

    // Display the tax amount
    System.out.printf("The tax amount for a salary of %.2f is: %.2f%n", salary, tax);
    // Close the scanner
    sc.close();
  }
}
```

OR

3. Explain the different types of operators in Java with examples. L2 CO1 10M

Java provides a wide range of operators to perform different types of operations on variables and values. Below are the main types of operators in Java along with examples:

1. Arithmetic Operators

These operators are used to perform basic mathematical operations.

Operator	Description	Example
+	Addition	a + b
-	Subtraction	a - b
*	Multiplication	a * b
/	Division	a / b
%	Modulus (remainder)	a % b

Example:

```
int a = 10, b = 5;
System.out.println(a + b); // Output: 15
System.out.println(a % b); // Output: 0
```

2. Relational (Comparison) Operators

These operators are used to compare two values.

Operator	Description	Example
==	Equal to	a == b
!=	Not equal to	a != b
>	Greater than	a > b
<	Less than	a < b
>=	Greater than or equal to	a >= b
<=	Less than or equal to	a <= b

Example:

```
int a = 10, b = 5;
System.out.println(a > b); // Output: true
System.out.println(a == b); // Output: false
```

3. Logical Operators

These operators are used to perform logical operations.

Operator	Description	Example
&&	Logical AND	(a > b) && (b > c)
!	Logical NOT	!(a > b)

Example:

```
int a = 10, b = 5, c = 15;
System.out.println((a > b) && (b < c)); // Output: true
System.out.println(!(a < b)); // Output: true
```

4. Assignment Operators

These operators are used to assign values to variables.

Operator	Description	Example
=	Assign	a = 10
+=	Add and assign	a += b
-=	Subtract and assign	a -= b
*=	Multiply and assign	a *= b
/=	Divide and assign	a /= b
%=	Modulus and assign	a %= b

Example:

```
int a = 10, b = 5;
a += b; // a = a + b
```

```
System.out.println(a); // Output: 15
```

5. Bitwise Operators

These operators perform operations on bits.

Operator	Description	Example
&	AND	a & b
`	`	OR
^	XOR	a ^ b
~	Complement	~a
<<	Left shift	a << 2
>>	Right shift	a >> 2

Example:

```
int a = 5, b = 3;
System.out.println(a & b); // Output: 1
System.out.println(a << 1); // Output: 10
```

6. Unary Operators

These operators operate on a single operand.

Operator	Description	Example
+	Unary plus	+a
-	Unary minus	-a
++	Increment	a++
--	Decrement	a--
!	Logical NOT	!a

Example:

```
int a = 10;
System.out.println(++a); // Output: 11
```

```
System.out.println(a--); // Output: 11 (then decrements to 10)
```

7. Ternary Operator

The ternary operator is a shorthand for an if-else condition.

Syntax:

```
condition ? expression1 : expression2;
```

Example:

```
int a = 10, b = 20;
int max = (a > b) ? a : b;
System.out.println(max); // Output: 20
```

8. Instanceof Operator

This operator checks whether an object is an instance of a specific class or subclass.

Example:

```
String str = \"Hello\";
System.out.println(str instanceof String); // Output: true
```

These operators provide powerful ways to manipulate data and control program flow in Java. Let me know if you need more details or additional examples!

UNIT – II

4. a) Differentiate constructor overloading and method overloading. Give suitable examples. **L3 CO2 5M**

Aspect	Method Overloading	Constructor Overloading
Definition	Method overloading occurs when a class has multiple methods with the same name but different parameters.	Constructor overloading occurs when a class has multiple constructors with different parameter lists.
Purpose	Increases the readability and reusability of the code.	Allows multiple ways to initialize an object with different sets of parameters.
Method Signature	Methods are differentiated by the number or type of parameters.	Constructors are differentiated by the number, type, or order of parameters.
Return Type	Methods may have any return type, including void.	Constructors do not have a return type.
Example of	By changing the number of arguments: void add(int a, int b) and void add(int a, int b, int	Constructor with different number of parameters: Person(String name) and

Aspect	Method Overloading	Constructor Overloading
Overloading Type	c). By changing the data type: void add(int a) and void add(double a)	Person(String name, int age)
Frequency of Use	More commonly used for methods performing similar tasks but with different input types or amounts.	Used to provide different ways to initialize an object based on the provided information.

Aspect	Method Overloading	Constructor Overloading
Method or Constructor Call	Overloaded methods can be called explicitly using their parameters.	Overloaded constructors are called during object creation using the new keyword.

Example Program	class Adder{
	static int add(int a, int b) {
	return a+b;
	}
	static double add(double a, double b) {
	return a+b;
	}
	}
	class TestOverloading2{
	public static void main(String[] args){
	System.out.println(Adder.add(11,11));
	System.out.println(Adder.add(12.3,12.6));}
	}
		public class Student {
		//instance variables of the class
		int id;
		String name;
		Student(){
		System.out.println("this a default constructor");
		}
		Student(int i, String n){
		id = i;
		name = n;
		}
		public static void main(String[] args) {
		//object creation
		Student s = new Student();
		System.out.println("\nDefault Constructor values: \n");
		System.out.println("Student Id : "+s.id + "\nStudent Name : "+s.name);
		System.out.println("\nParameterized Constructor values: \n");
		Student student = new Student(10, “ABC”);
		System.out.println("Student Id : "+student.id + "\nStudent Name : "+student.name);
		}
		}

b) Explain how you can modify string with example Program. L2 CO2 5M

In Java, string is an object that represents a sequence of characters.
Strings in Java are immutable, meaning once a String object is created, it cannot be changed. Any operation that tries to modify a string actually creates a new String object.

String methods to modify a string are:

1. String Concatenation:

This method combines a specific string at the end of another string and ultimately returns a combined string. **string concat()**

2. String Upper Case: The java string **toUpperCase()** method converts all the characters of the String to Upper Case.

3. String Lower Case: The java string **toLowerCase()** method converts all the characters of the String to lower case.

4. String substring(int startIndex): A part of string is called substring.This method is used to extract a portion of a string, starting from a specified index and continuing to the end of the string.

5. String substring(int startIndex, int endIndex):

This method in Java extracts a part of a string, starting from a specified startIndex and ending before a specified endIndex.

Example Program:

```
public class StringManipulation{
  public static void main(String[] args) {
    String s1 = "Hello";
    String s2 = "World";
    System.out.println("Original s1: " + s1); // Prints the original value of s1
    System.out.println("Original s2: " + s2); // Prints the original value of s2
    // String Concatenation using concat() method
    String combinedString2 = s1.concat(" ").concat(s2);
    System.out.println("Concatenation using concat(): " + combinedString2);
    // String to Upper Case
    String s1Upper = s1.toUpperCase(); // Converts s1 to uppercase
    System.out.println("\ns1 in uppercase: " + s1Upper);
    // String to Lower Case
    String s2Lower = s2.toLowerCase(); // Converts s2 to lowercase
    System.out.println("s2 in lowercase: " + s2Lower);
    // String substring from a starting index
    String sub1 = s1.substring(1); // Extracts a substring of s1 starting from index 1
    System.out.println("\nSubstring of s1 from index 1: " + sub1);
    // String substring from a starting index to an ending index
```

```
        String sub2 = s2.substring(0, 3); // Extracts a substring of s2 from index 0 up to (but not
including) index 3
        System.out.println("Substring of s2 from index 0 to 3: " + sub2);
    }
}
```

OR

5. a) Describe the process of declaring and initializing class and objects in Java with suitable example. **L2 CO2 5M**

class: A class is a group of objects which have common properties.It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

Declaration:

```
class <class_name>
{
fields(variables);
methods;
}
```

Initialization:

In Java, initialization of classes is generally done through **objects**, which are instances of the class.

```
ClassName objectName = new ClassName();
```

Object :

An entity that has state and behavior is known as an object. An Object in java essentially a block of memory that contains a space to store all the instance Variables.

Declaration:

```
classname objectname;
```

Initializing:

```
objectname=new classname();
```

We can directly define as:

```
classname objectname=new classname();
```

Example Prgram:

```
class Student // Declaration of Student class
{
   int id; // Declaration of id variable
   String name; // Declaration of name variable
}
class Example // Declaration of Example class
{
   public static void main(String args[])
   {
      Student s1 = new Student(); // Declaration and initialization of Student object
      s1.id = 101; // Initialization of id variable
      s1.name = "AAA"; // Initialization of name variable
      System.out.println(s1.id + " " + s1.name); // Printing id and name
   }
}
```

b) Construct a Java Program to differentiate passing arguments by value and by Reference. **L3 CO2 5M**

```
class Test {
   public int a;
   public int b;
   public Test(int a, int b) {
    this.a = a;
     this.b = b;
   }
}
public class SwapExample {
   // Method to swap primitive values (pass by value)
   public static void swap1(int a, int b) {
      int temp = a;
      a = b;
      b = temp;
   }
// Method to swap object values (pass by reference)
public static void swap2(Test t) {
   int temp = t.a;
   t.a = t.b;
   t.b = temp;
}
public static void main(String[] args) {
   int x = 10;
   int y = 20;
   System.out.println("Before Swapping using Pass by Value: x = " + x + ", y = " + y);
   // Pass by value
   swap1(x, y);
   System.out.println("After Swapping using Pass by Value: x = " + x + ", y = " + y);
   Test t = new Test(30, 40);
   System.out.println("Before Swapping using Pass by Reference: a = " + t.a + ", b = " + t.b);
   // Pass by reference
   swap2(t);
   System.out.println("After Swapping using Pass by Reference: a = " + t.a + ", b = " + t.b);
 }
}
```

UNIT – III

6. **a) List and explain different operations that can be performed on Array elements.** **L2 CO2 5M**

- **Traversing -** Iterating through array elements to access or display their values.
- **Inserting-**To insert an element at any index or at the end.
- **Deleting-** To delete an element at any index or at the end.
- **Searching-**To search for a specific element. It can be done using **binarySearch()** method.
- **Sorting-**To sort an array. It can be done **sort()** using method.
- **Comparing-** To check whether two arrays are equal or not. We can perform this by using **equals()** method.
- **Converting array into a string-**It can be achieved by using **toString()** method.

Example Program:

```
import java.util.Arrays;

import java.util.Scanner;

public class ArrayOperationsExample {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        System.out.println("Enter the size of the first array:");

        int size1 = scanner.nextInt();

        int[] numbers = new int[size1];

        System.out.println("Enter the elements of the first array:");

        for (int i = 0; i < size1; i++) {

            numbers[i] = scanner.nextInt();

        }

        System.out.println("Enter the size of the second array:");

        int size2 = scanner.nextInt();

        int[] array2 = new int[size2];

        System.out.println("Enter the elements of the second array:");

        for (int i = 0; i < size2; i++) {

            array2[i] = scanner.nextInt();

        }
```

// Insertion at 2nd position a value 10

```
int value = 10, position = 2;
// Copy elements
for (int i = 0, j = 0; i < array.length; i++)
{ if (i == position)
{
array2[i] = value;
}
else
{
Array2[i] = array2[j++];
}
```

// Deletion at 2nd position

```
int position = 2;

for (int i = position; i < array2.length - 1; i++)

{ array2[i] = array2[i + 1];

}
```

// Using for loop – Traversing

```
for (int i = 0; i < array2.length; i++)

{ System.out.println(array2[i]);

}
```

// Arrays.sort()

```
Arrays.sort(numbers); // Sorting the first array

System.out.println("Sorted first array: " + Arrays.toString(numbers));
```

// Arrays.binarySearch()

```
System.out.println("Enter a value to search in the sorted first array:");

int searchValue = scanner.nextInt();

int index = Arrays.binarySearch(numbers, searchValue); // Binary search for the user
input value

System.out.println("Index of " + searchValue + ": " + index);
```

// Arrays.equals()

```
        boolean areEqual = Arrays.equals(numbers, array2); // Check if arrays are equal
        System.out.println("Are the first and second arrays equal? " + areEqual);

// Arrays.fill(): Fills the second array with the specified value.
        System.out.println("Enter a value to fill the second array:");
        int fillValue = scanner.nextInt();
        Arrays.fill(array2, fillValue); // Fill the second array with the specified value
        System.out.println("Filled second array: " + Arrays.toString(array2));
        // Arrays.toString()
        String arrayString = Arrays.toString(numbers); // Convert the first array to a string
representation
        System.out.println("First array as string: " + arrayString);
        scanner.close();
    }
}
```

b) What is inheritance? Explain different types of inheritance techniques which are supported by Java. **L2 CO2 5M**

Inheritance: Inheritance is a concept in object-oriented programming (OOP) where a class (called a subclass or derived class) inherits the properties and behaviours (methods) from another class (called a superclass or base class). This allows for the reuse of code, enabling new classes to be built upon existing ones.

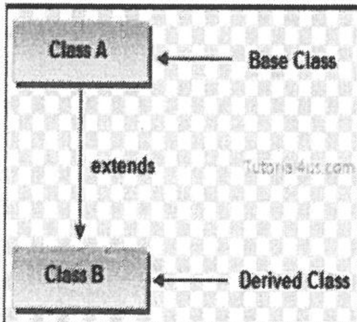
A Class that is Inherited(Old Class) is called a "Super Class" or "Base Class" or "Parent Class".

A Class that does the inheriting(New Class) is called a "Sub Class" or " Derived Class" or "Child Class".

In java only Three types of Inheritances supported .**These are single, multi level, hierarchical.**

Single Inheritance: It is a type of inheritance in Java where a subclass (child class) inherits the properties and behaviours (fields and methods) from a single superclass (parent class). This allows the subclass to reuse, override, or extend the functionality of the superclass. And

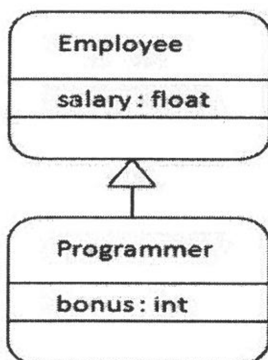
establishes a relationship between different classes, making it easier to maintain and extend code.



Syntax:

```
class Parent {
   // Parent class code
}
class Child extends Parent {
   // Child class code
}
```

Example:



Multilevel Inheritance: It is a type of inheritance in Java where a class derives from a class that is itself derived from another class. This creates a chain of inheritance, allowing properties and behaviors to propagate through multiple levels of classes.

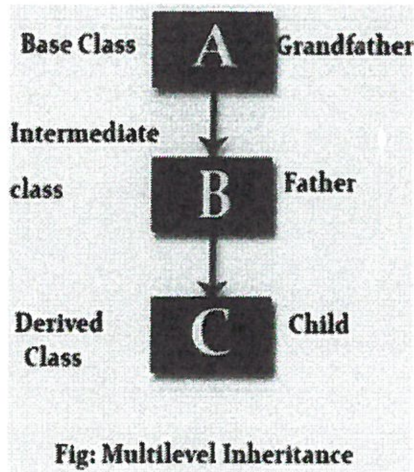
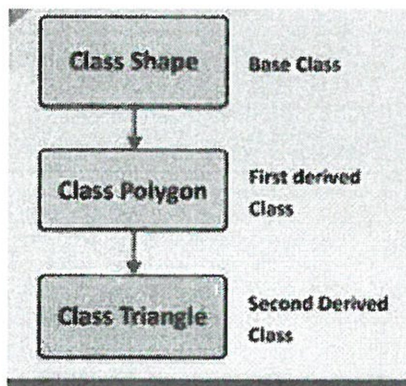


Fig: Multilevel Inheritance

Syntax:

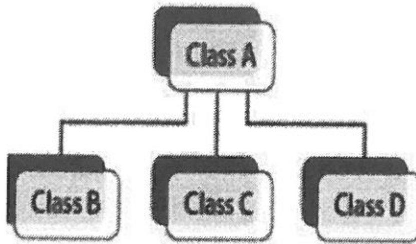
```
class GrandParent {
   // Grandparent class code
}
class Parent extends GrandParent {
   // Parent class code
}
class Child extends Parent {
   // Child class code
}
```

Example:



Hierarchical Inheritance: This inheritance in Java is a type of inheritance where a single parent class is extended by multiple child classes. Each child class inherits the properties and methods of the parent class but can also define its own unique features.

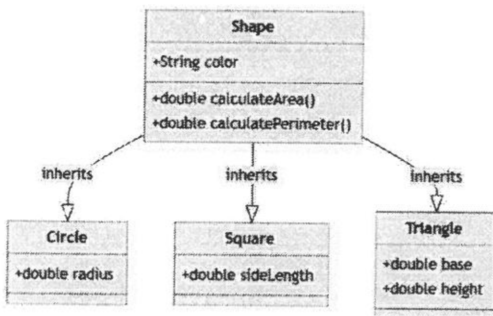
In the below image, class A serves as a base class for the derived classes B, C, and D.



Syntax:

```
class A {
   // A class code
}
class B extends A {
   // B class code
}
class C extends A {
   // C class code
}
class D extends A {
   // D class code
}
```

Example:



OR

7. a) Explain the following **L2 CO2 5M**
i) final keyword in inheritance **ii) vector**

i) final keyword in inheritance:

The final keyword in java is used to restrict the user. The java final keyword can be used in many context. final can be: 1. variable 2. method 3. class

The main purpose of using a class being declared as final is to prevent the class from being sub classed. If a class is marked as final then no class can inherit any feature from the final class.

If you make any class as final, you cannot extend it, which means a class which is made final cannot be inherited.

```
final class Bike{  }

class Honda extends Bike{

void run(){System.out.println("running safely with 100km ph");

}

public static void main(String args[]){

Honda  honda= new Honda();

honda.run();

}

}
```

Note: the above program gives "Compile time error" Since class Bike is declared as final so the derived class Honda cannot extend Bike

ii) Vector:

Vectors are another kind of data structure that is used for storing information. Using vector, we can implement a dynamic array. Vectors are dynamically allocated.

The vector class is contained in java.util package. Vector stores pointers to the objects and not objects themselves.

The following are the vector constructors.

```
Vector vec = new Vector( 5 );    // size of vector is
```

Example Program:

```
import java.util.Vector;

public class VectorExample {

    public static void main(String[] args) {

        Vector<Integer> vector = new Vector<>();

        // Add elements dynamically
```

```
        vector.add(10);
        vector.add(20);
        vector.add(30);
        // Access elements
        System.out.println("Element at index 1: " + vector.get(1));
    }
}
```

b) Discuss the concepts of default and static methods in interface. **L2 CO2 5M**

In java, interfaces can contain **default methods** and **static methods.** Before Java 8, interfaces could have only abstract methods. The implementation of these methods has to be provided in a separate class. So, if a new method is to be added in an interface, then its implementation code has to be provided in the class implementing the same interface. To overcome this issue, Java 8 has introduced the concept of default methods which allow the interfaces to have methods with implementation without affecting the classes that implement the interface

Default Method:

A default method is defined inside an interface using the default keyword followed by the method body. It is a concrete method, meaning it has a body unlike abstract methods which only have a declaration.

Syntax: The default keyword is used before the method declaration in the interface.

Example: default void defaultMethod() ;

Static Method:

A **static method** in an interface is similar to static methods in classes. Static methods in interfaces are not inherited by implementing classes, and they must be called on the interface itself rather than on an object of the class implementing the interface.

A static method in an interface is defined with the static keyword. It can have a method body (like any other static method) and is not inherited by classes that implement the interface.

Syntax: The static keyword is used to define the method in the interface.

Example: static void Staticmethod();

Example program:

```
interface Drawable {
```

```
    void draw();
    default void message() {
        System.out.println("This is a Drawable shape.");
    }
    static void printInfo() {
        System.out.println("Drawable interface provides drawing functionality.");
    }
}
class Rectangle implements Drawable {
    public void draw() {
        System.out.println("drawing rectangle");
    }
}
class Circle implements Drawable {
    public void draw() {
        System.out.println("drawing circle");
    }
}
class Test {
    public static void main(String args[]) {
        Drawable d = new Circle();// In real scenario, object is provided by method e.g.
getDrawable()
        d.draw();
        d.message(); // Calling default method
        Drawable.printInfo(); // Calling static method
        d = new Rectangle();
        d.draw();
        d.message();// Calling default method
```

```
    }
}
```

UNIT – IV

8. a) How do different access control specifiers control access to class members across different packages? Explain. **L2 CO3 5M**

In Java, **access control specifiers** (also known as **access modifiers**) control the visibility and accessibility of class members (fields, methods, and inner classes) across different packages. Java provides four access control specifiers: public, protected, default (package-private), and private. These specifiers determine how accessible a class or its members are from other classes, including those in different packages.

The access specifiers are public, private, protected and default

a) **Public:** The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

Program:

```
//save by A.java

package pack;

public class A{

public void msg(){System.out.println("Hello");}

}

//save by B.java

package mypack;

import pack.*;

class B{

public static void main(String args[]){

A obj = new A();

obj.msg();
```

```
}
}
```

Output:

Hello

b) **Private:** The access level of a private modifier is only within the class. It cannot be accessed from outside the class. private members are not accessible from any other class, whether it is in the same package or a different package.

Program:

Save A.java

```
package mypackage;
public class A {
    private int data = 40; // Private field
    private void msg() {   // Private method
        System.out.println("Hello java");
    }
}
```

Save Simple.java

```
import mypackage.A; // Importing the class from 'mypackage'
public class Simple {
    public static void main(String[] args) {
        A obj = new A();
        // Attempt to access private members
        System.out.println(obj.data); // Compile Time Error
        obj.msg();                    // Compile Time Error
    }
}
```

Output:

Compile Time Error

c) **Protected:** The access level of a protected modifier is within the package and outside the package through child class. If we do not make the child class, it cannot be accessed from outside the package.
In a different package, protected members can only be accessed if the class is a **subclass** (either directly or indirectly) of the class that defines the protected member. Non-subclasses in different packages cannot access protected members.

```
//save by A.java

package pack;

public class A{

protected void msg(){System.out.println("Hello");}

}

//save by B.java

package mypack;

import pack.*;

class B extends A{

public static void main(String args[]){

B obj = new B();

obj.msg();

}

}
```

Output:

Hello

d) **Default:** The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If we do not specify any access level, it will be the default.

Program:

```
//save by A.java

package pack;

class A{

void msg(){System.out.println("Hello");}
```

```
}
//save by B.java
package mypack;
import pack.*;
class B{
public static void main(String args[]){
A obj = new A();//Compile Time Error
obj.msg();//Compile Time Error
}
}
```

Output:

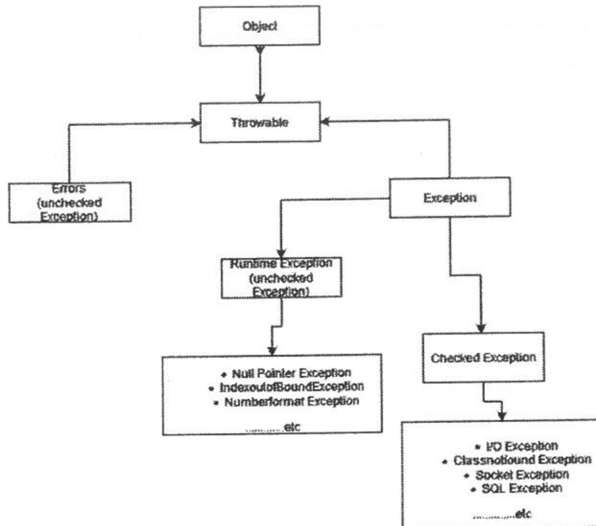
Compile Time Error

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

b) Differentiate between checked and unchecked exceptions. Give suitable examples. **L3 CO3 5M**

An exception is an abnormal condition that arises in a code sequence at run time. Exception is a run time error. Java and other programming languages have mechanisms for handling exceptions that you can use to keep your program from crashing. In Java, this is known as catching an exception/exception handling.

Hierarchy of Standard Exception Classes:



There are two types of exceptions in Java:

• Unchecked Exceptions • Checked Exceptions

Differences:

Aspect	Checked Exceptions	Unchecked Exceptions
Definition	Exceptions checked (notified) by the compiler at compilation time.	Exceptions that occur during runtime and are not checked at compilation time.
Alternative Name	Compile-Time Exceptions	Runtime Exceptions
Detection	Detected by the compiler during the code compilation process.	Detected only when the program is executed.
Error Type	Most are caused by syntax errors like missing semicolons, brackets, or misspelled identifiers.	Typically caused by programming bugs, such as logic errors or improper API usage.
Impact on Compilation	Compiler displays errors and stops generating the bytecode file until the errors are resolved.	Program compiles successfully (bytecode is generated), but may crash or behave unexpectedly during execution.
Examples	- **ClassNotFoundException**: Class not found. - **IllegalAccessException**: Access to a class is denied. - **NoSuchMethodException**: A requested method does not exist. - **InterruptedException**: A thread is interrupted by another thread.	- **ArithmeticException**: Arithmetic error, such as division by zero. - **ArrayIndexOutOfBoundsException**: Accessing an invalid index in an array. - **NullPointerException**: Invalid use of a null reference. - **NumberFormatException**: Invalid conversion of a string to numeric format.

OR

9. **a) Explain the usage of try, catch, throw, throws and finally keywords in exception handling. Give simple example.** **L2 CO3 5M**

1. **try block :**

The statements that are produces exception are identified in the program and the statements are placed with within a try block. If an Exception occurs with in the try block, the appropriate exception handler(catch block) associated with try block handles the Exception Immediately.

2. **catch block :**

The catch block is used to process the Exception raised. The catch block is placed immediately after the try block.

```
try{
// Block of code to try
}
catch(Exception e) {
// Block of code to handle errors
}
```

3. **finally block :**

The finally block follows a try block or a catch block.Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

```
try {
   // Code that might throw an exception
} catch (ExceptionType1 e1) {
   // Handling specific exception type 1
} catch (ExceptionType2 e2) {
   // Handling specific exception type 2
} finally {
   // Code that will always execute, regardless of exception
}
```

4. **throw statement:**

It is also possible to create a program that throws an exception explicitly, using the "throw" statement.

Syntax: throw new ExceptionType("Error message");

5. throws statement:

The Java throws keyword is used to declare an exception.It gives an information to the programmer that there may occur an exception.so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.

Syntax: returnType methodName() throws ExceptionType1, ExceptionType2 {

```
// Method implementation
}
```

Example Program:

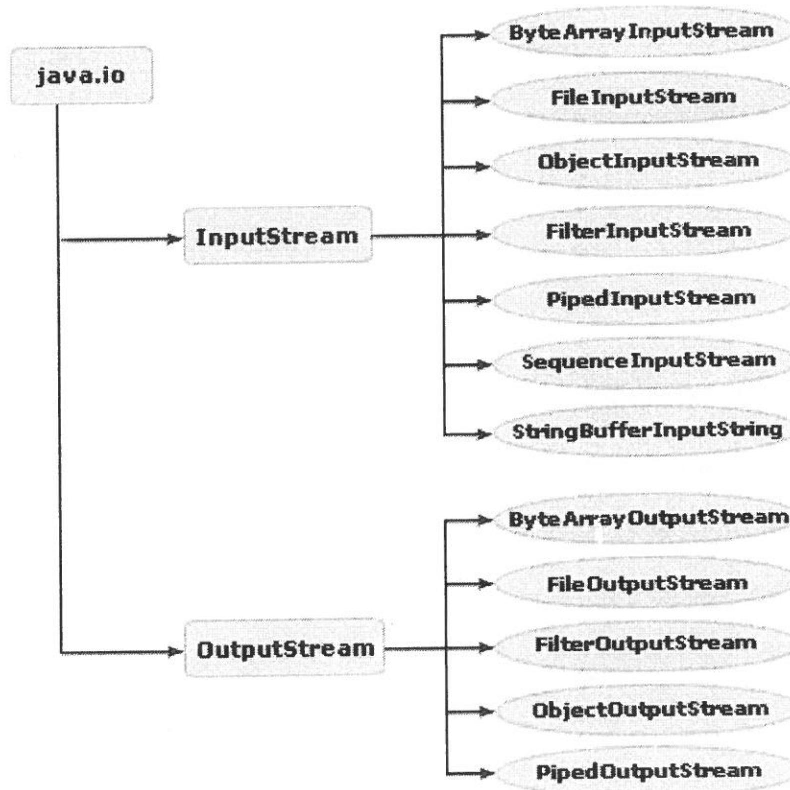
```
class t {
  public t() throws NullPointerException {
    System.out.println("caught");
    throw new NullPointerException("demo");
  }
}
class th3 {
  public static void main(String args[]) {
    try {
      t obj = new t();
    } catch (NullPointerException e) {
      System.out.println("Recaught");
    } finally {
      System.out.println("Finally block executed");
    }
    System.out.println("rest of code");
  }
}
```

```
public class SimpleExceptionExample {

    // Method that may throw an exception
    public static void divide(int a, int b) throws ArithmeticException {
        if (b == 0) {
            throw new ArithmeticException("Cannot divide by zero");
        }
        System.out.println("Result: " + (a / b));
    }

    public static void main(String[] args) {
        int num1 = 10, num2 = 0;
        try {
            // Attempt to divide two numbers
            divide(num1, num2);
        } catch (ArithmeticException e) {
            // Catch the exception if it occurs
            System.out.println("Caught exception: " + e.getMessage());
        } finally {
            // Finally block that always runs
            System.out.println("This will always be executed, even if there's an exception.");
        }

        System.out.println("Program finished.");
    }
}
```

b) Illustrate the difference between byte stream and character streams in Java. Draw the stream hierarchies. **L3 CO3 5M**

Aspect	Character Streams	Byte Streams
Purpose	Specialized for handling text data (Unicode characters).	Used to process raw binary data.
Inheritance	Classes inherit from Reader (for input) and Writer (for output).	Classes inherit from InputStream (for input) and OutputStream (for output).
Common Classes	- FileReader - FileWriter	- FileInputStream - FileOutputStream
Data Type Handled	Deals with character data, supports Unicode.	Deals with byte data, handles raw binary data (e.g., images, audio).
Encoding	Automatically handles character encoding and decoding.	Does not handle character encoding, works with raw bytes.
Use Case	Used for reading/writing text files, such as .txt files.	Used for reading/writing binary files, such as images, audio files.
Example Usage	- FileReader: Reads characters from a file. - FileWriter: Writes characters to a file.	- FileInputStream: Reads bytes from a file. - FileOutputStream: Writes bytes to a file.

Stream Hierarchy:



UNIT – V

10. a) Explain how thread priority is set and used in Java with example program. **L4 CO4 5M**

When the threads are created and started, a "thread scheduler" program in JVM will load them into memory and execute them. This scheduler will allot more JVM time to those thread which are having priority. The priority numbers will change from 1 to 10.

Thread.MAX_PRIORITY – 10

Thread.MIN_PRIORITY – 1

Thread.NORM_PRIORITY – 5

Setting Thread Priority: You can set a thread's priority using the **setPriority(int priority)** method, where the priority is an integer between 1 and 10

Program:

```
class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + " with priority: " +
            getPriority());
    }
}
public class ThreadPriorityExample {
    public static void main(String[] args) {
        MyThread thread1 = new MyThread();
        MyThread thread2 = new MyThread();
        MyThread thread3 = new MyThread();
        // Setting priorities
        thread1.setPriority(Thread.MAX_PRIORITY); // Highest priority (10)
        thread2.setPriority(Thread.NORM_PRIORITY); // Default priority (5)
```

```
        thread3.setPriority(Thread.MIN_PRIORITY); // Lowest priority (1)

        // Starting threads

        thread1.start();

        thread2.start();

        thread3.start();

    }

}
```

b) Illustrate the use of HashSet class in collection framework with an example program. **L3 CO4 5M**

Creating a HashSet: import java.util.HashSet;

```
HashSet numbers = new HashSet<>(8, 0.75);
```

The first parameter is capacity, and the second parameter is loadFactor.

Capacity - The capacity of this hash set is 8. Meaning, it can store 8 elements.

LoadFactor - The load factor of this hash set is 0.6. This means, whenever our hash set is filled by 60%, the elements are moved to a new hash table of double the size of the original hash table.

```
// HashSet with default capacity and load factor

HashSet numbers1 = new HashSet<>();
```

Uses of Hashset:

1.A HashSet is designed to store only unique elements. If you try to add a duplicate element, it will not be added.

2. HashSet does not guarantee any specific order of the elements (no insertion order or sorted order), it is useful when you don't need to maintain order.

3.If you need to ensure that a set of data is processed only once (e.g., reading unique data), a HashSet is ideal to avoid redundant operations.

Program:

```
import java.util.*;

public class SetExample {

    public static void main(String[] args) {

        // Create a HashSet to store unique strings
```

```
        Set<String> set = new HashSet<>();
        // Create a scanner object to take user input
        Scanner scanner = new Scanner(System.in);
        System.out.print("How many elements would you like to enter? ");
        int numElements = scanner.nextInt();
        scanner.nextLine(); // Consume the newline character
        // Use a for loop to get a fixed number of inputs
        for (int i = 0; i < numElements; i++) {
            System.out.print("Enter element " + (i + 1) + ": ");
            String input = scanner.nextLine();
            // Add the input to the set (duplicates will be ignored)
            set.add(input);
        }
        // Close the scanner
        scanner.close();
        // Output the final set of unique elements
        System.out.println("Set contains: " + set);
    }
}
```

OR

11. a) Analyze different procedures for creating a thread in Java. Explain any one mechanism with example program. **L4 CO4 5M**

Ans:A Thread is a very light-weighted process, or we can say the smallest part of the process that. allows a program to operate more efficiently by running multiple tasks simultaneously.

We can create Threads in java using two ways, namely :

1. Extending Thread Class

2. Implementing a Runnable interface

By Extending Thread Class:

We can run Threads in Java by using Thread Class, which provides constructors and methods for

creating and performing operations on a Thread, which extends a Thread class that can implement

Runnable Interface. We use the following constructors for creating the Thread:

Thread

Thread(Runnable r)

Thread(String name)

Thread(Runnable r, String name)

*example:Sample code to create Threads by Extending Thread Class:

```
import java.io.*;
import java.util.*;
public class GFG extends Thread {
// initiated run method for Thread
public void run()
{
System.out.println("Thread Started Running...");
}
public static void main(String[] args)
{
GFG g1 = new GFG();
// Invoking Thread using start() method
g1.start();
}
}
```

2.

*example:Sample code to create Thread by using Runnable Interface:

```
import java.io.*;
import java.util.*;
public class GFG implements Runnable {
// method to start Thread
public void run()
{
System.out.println(
"Thread is Running Successfully");
}
public static void main(String[] args)
{
GFG g1 = new GFG();
// initializing Thread Object
Thread t1 = new Thread(g1);
t1.start();
}
}
```

b) Define List and differentiate ArrayList, Linked List. **L2 CO4 5M**

List: An ordered collection that allows duplicates (e.g., ArrayList, LinkedList).

Feature	ArrayList	LinkedList
Data Structure	Uses a dynamic array to store elements.	Uses a doubly linked list (nodes with pointers).
Implementation	Implements the List interface.	Implements the List and Deque interfaces.
Order Maintenance	Maintains insertion order.	Maintains insertion order.

Feature	ArrayList	LinkedList
Duplicates	Allows duplicate elements.	Allows duplicate elements.
Access Type	Provides random access to elements using indices.	Provides sequential access (linked node traversal).
Memory Usage	More memory-efficient because it stores elements in contiguous memory locations.	Uses more memory due to storage of additional pointers (prev/next).
Resize Operation	ArrayList grows dynamically when the array is full, but resizing can be costly.	No resizing required as it grows or shrinks by adding/removing nodes.
Use Cases	Ideal for scenarios where random access to elements is needed.	Ideal for scenarios where frequent insertions/deletions are needed.

Array List example program

```
import java.util.*;
class TestJavaCollection1{
public static void main(String args[]){
ArrayList<String> list=new
ArrayList<String>();//Creating arraylist
list.add("A");//Adding object in arraylist
list.add("B");
list.add("C");
list.add("A");
//Traversing list through Iterator
Iterator itr=list.iterator();
while(itr.hasNext()){
System.out.println(itr.next()); }
}
}
```

Linked list example program:

```
import java.util.*;
class TestJavaCollection2 {
   public static void main(String args[]) {
      // Creating LinkedList
      LinkedList<String> list = new LinkedList<String>();
      // Adding objects to LinkedList
      list.add("A");
      list.add("B");
      list.add("C");
      list.add("A");
      // Traversing list through Iterator
      Iterator<String> itr = list.iterator();
      while (itr.hasNext()) {
         System.out.println(itr.next());
      }
   }
}
```